Trends in Compiler Construction
Invited Address

BRUCE W. WATSON
Faculty of Mathematics and Computing Science
Eindhoven University of Technology
5600 MB, Eindhoven, The Netherlands
E-Mail: watson@win.tue.nl
Facsimile: +31 40 436685

Abstract: Compiler writing techniques have undergone a number of major revisions over the past forty years. The introduction of object-oriented design and implementation techniques promises to improve the quality of compilers, while making large-scale compiler development more manageable.

In this paper, we will show that a new way of thinking of a compiler's structure is required to achieve complete object-orientation. This new view on compiling can lead to alternative formulations of parsing and code generation. In practice, the object-oriented formulations have not only proven to be highly efficient, but they have also been particularly easy to teach to students.

Keywords: Object-oriented compilers, parsers, object-oriented parsers

Computing Review Category (1991): D.3.4 [**Programming Languages**] Processors — *compilers; parsing*

1 Introduction

The craft of compiler construction has undergone some major changes since the first compilers were developed in the late 1950s. Due to the lack of a clear understanding of the theoretical issues involved in compilation, early compilers were constructed in a rather disorderly way. Although the organization of the data-structures and the phases of a compiler left much to be desired, the relatively small compilers did not create significant correctness or code maintenance problems. The first major improvement came when compiler writers in the 1960s and early 1970s concentrated on the use of fundamental new algorithms (such as LR parsing and tree pattern matching) to give the craft a more solid foundation. From the mid 1970s to the mid 1980s, the quality of compilers was greatly improved upon with the introduction of various compiler component generators (implementing these newly discovered algorithms).

Until the mid 1980s, compilers were largely batch programs, reading a source file and producing an object file. In the late 1980s and early 1990s, many new compiler variants appeared in the marketplace, for example: integrated environments, global optimizers, retargetable compilers, and incremental compilers.

The complexity of the new generation of compilers has lead to problems: compilers are difficult to implement correctly, the software engineering issues (for a family of compilers) are difficult to manage, and compiler architecture is becoming difficult to teach to students. To counter the

growing unmanageability of compiler development, object-oriented (O-O) methods are now being introduced into compiler construction. From other areas of software engineering, O-O methods (in particular, correctness-oriented methods such as *design by contract* [3]) are known to yield significant improvements in the quality and maintainability of code. Unfortunately, even the most recent book on compiler construction [2] does nothing more than present auxiliary data-structures (such as symbol tables) in an O-O manner, while retaining the traditional procedural approach (including Lex and YACC) for lexing and parsing. In this paper, we will consider a truly O-O approach to compiler design — recursive descent parsing and LR parsing will both be presented in a purely O-O manner.

This paper is organized as follows:

- Section 2 gives an overview of the structure of an object-oriented compiler.
- Parsing is described in an object-oriented way in Section 3.
- The conclusions of this paper are given in Section 4.

Throughout this paper, we assume that the reader has a basic grasp of compilers and their construction.

2 Object-orientation and compilers

A traditional compiler design consists of a number of *phases,* for example, scanning, parsing, and attribute evaluation. Such a phase organization is usually drawn as a diagram as in Figure 1. A great deal of research has gone into separating the phases. The procedure-call interface to each phase can be rigorously defined and agreed upon, allowing the implementor to code and test each phase separately. An excellent example of this method of compiler writing is given in [1].

Current attempts at the construction of object-oriented compilers are half-hearted. The application of O-O techniques is limited to a few parts of the compiler, such as the symbol table and the parse tree, while the rest of the compiler is implemented procedurally. Such mixed paradigm programming can create a number of difficulties:

- Mixed paradigm designs are difficult to specify, since they require a mix of formalisms.
- Programmers have difficulty switching between the procedural and O-O programming.
- Some of the best O-O languages, such as Eiffel, do not support procedural programming at all.

Little attention is usually paid to the data-structures that are passed between phases of the compiler. The O-O design of a compiler requires us to think in terms of the data-structures between the phases of the compiler (e.g. token-stream, parse tree, and program dependence graph). Figure 2 shows the organization of such a compiler. The data-structures appear more prominently, with the phases now being incidental. Once the data-structures are designed (as classes), the phases of the compiler arise naturally as the constructors of each of the classes. For example, a parse tree object knows how to construct itself from a token-stream. Contrast this with the traditional approach, in which a (global) parser function takes a token stream and constructs a parse tree.

In the following section, we consider part of an O-O compiler: the parse tree and parser.

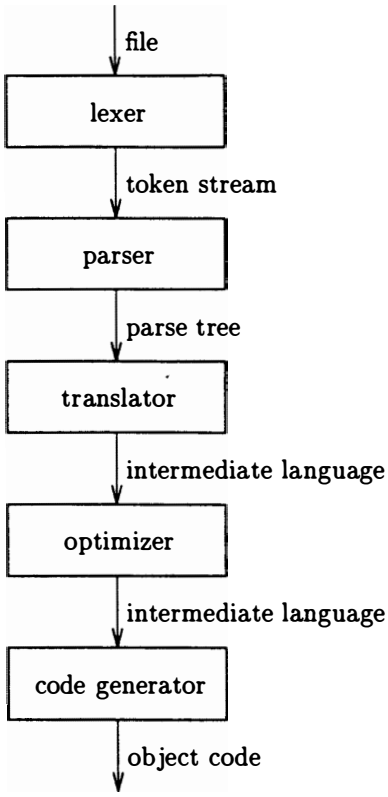

Figure 1: Traditional compiler organization as phases.

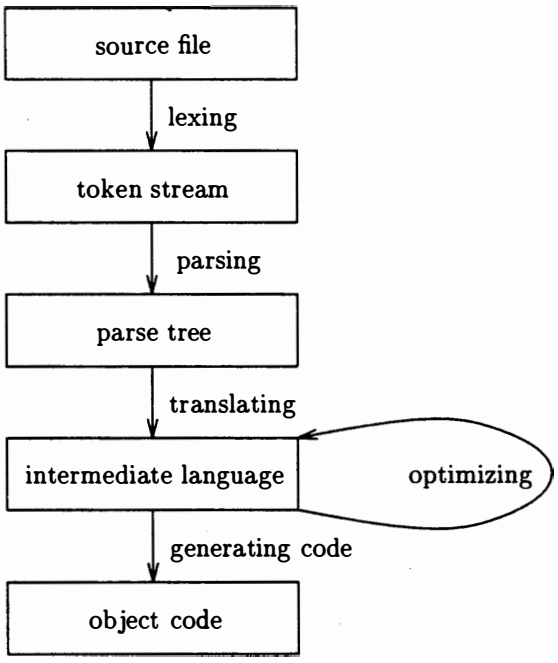

Figure 2: An object-oriented compiler organization.

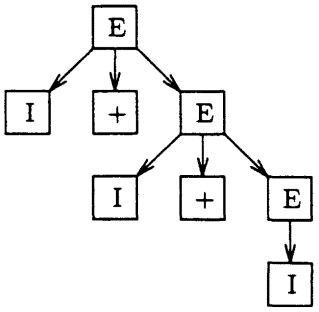

Figure 3: Parse tree built top-down.

3 Object-oriented parsing

A naïve approach to O-O parsing (and the approach most often taken) is to create a parser class. This class then has a member function which constructs the parse tree, given a token-stream. This approach is little more than an O-O wrapper around a traditional procedural parser and it does little to achieve the benefits of fully O-O design. The approach is used in [2]. We now consider true O-O parsing.

In order to consider O-O parsing, we begin by assuming that we have already designed the token-stream class and the classes required to represent the parse tree. The parsing algorithm itself will be embodied in the constructors of the parse tree nodes. We will consider both top-down and bottom-up parsing in Sections 3.1 and 3.2 respectively.

Some of the newest programming languages (for example, C++) are poorly designed in that they require a back-tracking parser to correctly process a source program. Back-tracking in an O-O parser is considered in Section 3.3.

3.1 Top-down parsing

Top-down parsing is the most easily understood method of constructing the parse tree. We begin by constructing an object representing the root of the parse tree, passing the token stream to the constructor. The token stream is then used to construct the subtrees of the root. To present an example, we introduce a very simple expression grammar (where E is for *expression* and I is for *identifier*):

$$\begin{array}{rcl} E & \longrightarrow & I + E \\ & | & I \end{array}$$

(For simplicity, we do not elaborate on whether I is a terminal symbol, or a nonterminal.) In our example, the constructor for E invokes the constructor for I (passing it the token stream), constructing a new I object. (In the case that I is a token, the I will simply be extracted from the token stream instead of being constructed.) The I constructor would then use the token stream to build its subtree. The E constructor then determines if the next token is a +; if it is, the constructor constructs a new E object, again passing it the token stream. The parse tree of string $I + I + I$ is shown in Figure 3. When the parse tree is not needed after the parsing of the input[1], a node can be safely destroyed after the successful invocation of the constructors of its subtrees. In this case, the class itself can be eliminated, making the constructor operate as a standalone procedure. This yields the classic *recursive descent* parsing scheme.

[1] This can happen in the case where the compiler is only checking syntax.

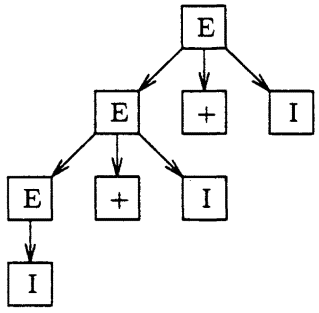

Figure 4: Parse tree built bottom-up.

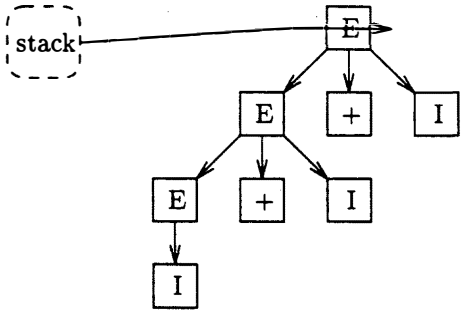

Figure 5: Parse tree with root member invoked.

The class definitions and their constructors can easily be generated using a type of parser generator which takes the grammar as input. Such a tool has been written for use in [4].

3.2 Bottom-up parsing

Bottom-up parsing is known to be more complex and more powerful than top-down parsing. In this section, we present a non-rigorous (but easily understood) description of bottom-up parsing. We will make use of the following version of the original grammar:

$$\begin{array}{rcl} E & \longrightarrow & E + I \\ & | & I \end{array}$$

The same input as before, $I + I + I$, yields the parse tree in Figure 4. Let us first consider how to *un-parse* the tree into a stream of tokens, starting with the right-most leaf. One possible strategy is the following: we invoke a member function of the root E object; this is shown in Figure 5, where the horizontal arrow represents the procedure/function invocation stack — the member function of the root is the currently active function[2]. This member function determines that each of the object's children must be un-parsed in-turn. It therefore destroys the root object, returning enough information (to $\perp$) so that the un-parse member function of its children are invoked from left to right (each of them simply invokes the next sibling, until the rightmost sibling). The act of passing control to the children is known as a *produce* step. This leaves us with the run-time situation depicted in Figure 6. The currently active member function, that for the rightmost I

[2]There may be other active functions on the stack — indeed we will assume that the second to the top of the stack is a special function called $\perp$, whose invocation is not shown.

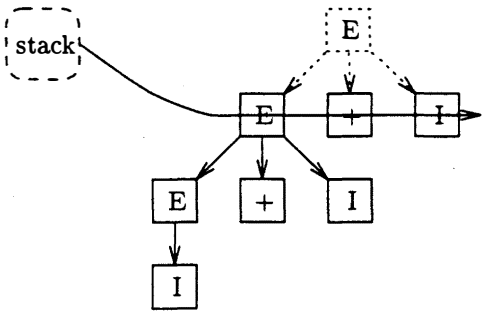

Figure 6: Parse tree with first level children invoked.

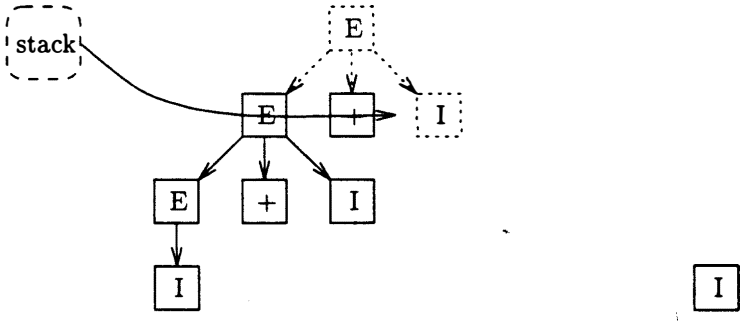

Figure 7: Parse tree after a shift step.

object, outputs the I object and returns. This yields Figure 7, in which the member of the + is active and the I node is shown to the right as output. This is known as a *shift* step. The + node also performs a shift step, yielding Figure 8. The currently active E node member function determines that the node's children must un-parse themselves. As with the original root of the tree, the member returns enough information for the children's member functions to be invoked — another produce step — yielding Figure 9. Figures 10, 11, 12, and 13 show the situation after two shifts, a produce, and a shift, respectively. The last figure shows the parse tree complete decomposed into the token stream $I + I + I$. (The stack indicates that the $\perp$ function is active.)

The bottom-up parsing process is the time-reversal of the above un-parse process. The reverse of a *produce* step is a *reduce* step, while a *shift* step is still called a *shift* step. As such, the sequence of figures should be read backwards. While the un-parsing process is obviously deterministic, it is possible that the time-reversal is not deterministic. The parsing process can be made deterministic by introducing one or more tokens of lookahead (e.g. LR(1)), or by restricting the class of grammars (e.g. LR(0)).

This parsing method is essentially an O-O approach to LR parsing. The O-O version has a number of advantages over traditional LR parsing:

- This particular method of presentation has been tried on students, most of whom find it much more understandable than the traditional "items, handles, stacks, and tables" presentation.
- It is much faster in practice.
- The use of member functions eliminates the need for an artifically maintained stack, as in traditional LR parsers.

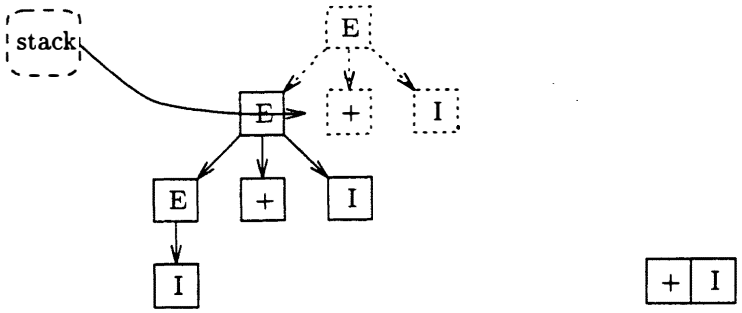

Figure 8: Parse tree after a shift by the + node.

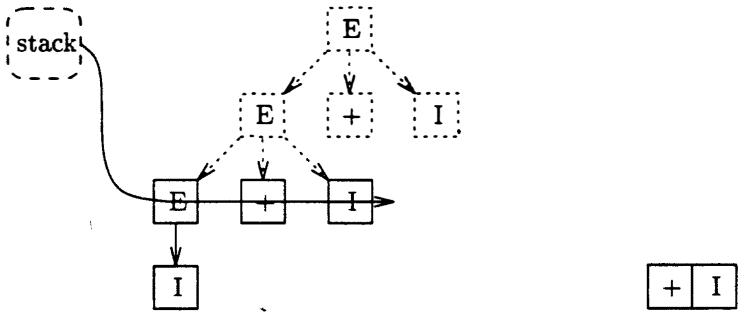

Figure 9: Parse tree after another produce step.

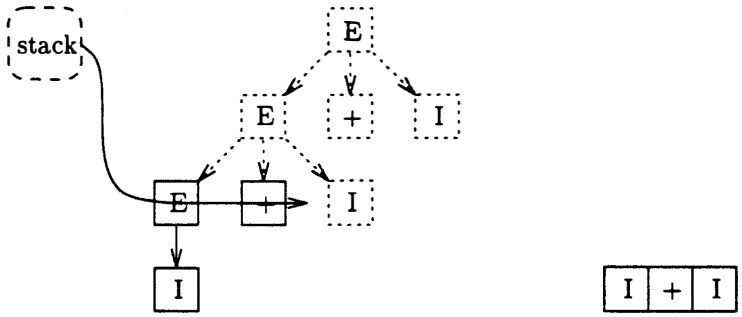

Figure 10: Parse tree after a shift of the middle I.

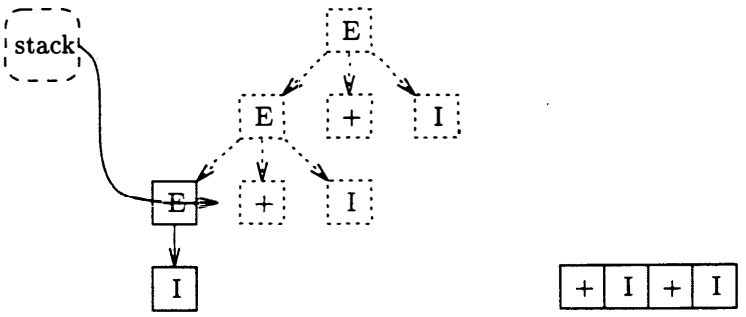

Figure 11: Parse tree after the shift of the left +.

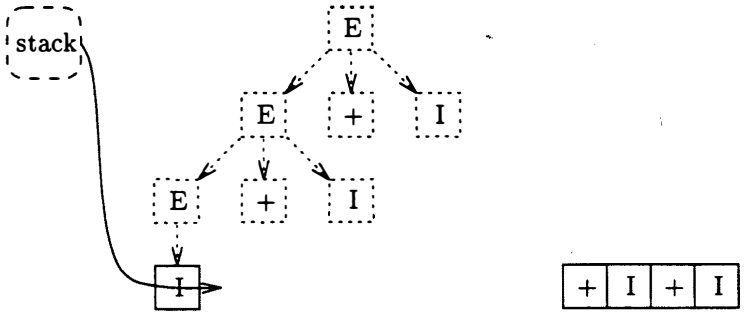

Figure 12: Parse tree after the produce of the left E.

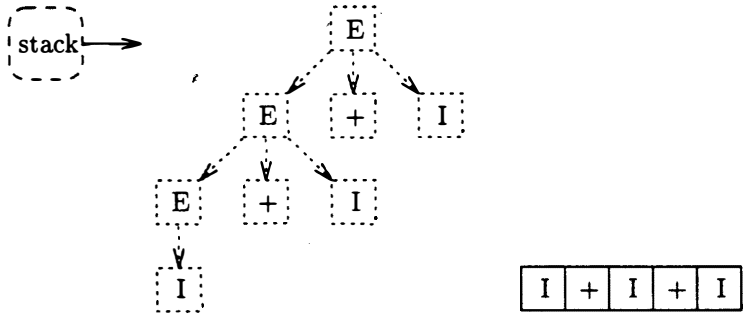

Figure 13: Parse tree after the last shift.

- The parse tree is constructed as a matter of course, without having to do it manually.
- The shift-reduce information is distributed to the relevant places in member functions of classes (representing the tokens and nonterminal nodes). It is even possible to write and debug such an O-O LR parser by hand.

While such parsers can be written by hand, a parser generator has been written (for [4]) which creates all of the required class and constructor definitions.

3.3 Backtracking in parsers

For some grammars there are strings which a parser cannot parse correctly in a completely deterministic manner. (Unfortunately, the draft standard C++ grammar is an example.) For such grammars, we would like to use a backtracking parser: one which attempts to parse the string, backing up (to try alternatives) when it gets stuck.

In our O-O parser, we may have constructed part of the parse tree before we discover the need to backtrack. The part that is already constructed will depend upon the type of parser being used: in a top-down parser the upper part of the tree will have been built, while in a bottom-up parser the lower part will have been built. Backtracking requires us to be able to make a non-local jump to the point at which we will make the next attempt at parsing; during this non-local jump, part of the parse tree (the part corresponding to the failed parsing attempt) must be destroyed.

Such non-local transfers of control (to some enclosing block in the program, including the destruction of data-structures) can be easily dealt with using exceptions. As soon as the need for backtracking is discovered (in the parser), the parser raises an exception. We place an exception handler (a *catcher* in C++) at the point in the parser where an alternative parsing attempt is to be made. The handler will catch the exception and initiate the alternative attempt. Exceptions are usually reserved for truly exceptional failures of a program. In practice, using them for backtracking parsing has proven to be much more efficient than achieving the same effect without them.

4 Conclusions

The following conclusions can be drawn about the current state of O-O compiler research and development:

- To effectively apply O-O compiler design and development requires a different view of the compilation process.
- O-O techniques assist in managing the complexity of sofware design and implementation. This is especially true in compiler development, where a family of compilers can require a massive implementation effort.
- O-O techniques such as *design by contract* [3] can assist greatly in creating correct compilers.
- O-O compilers can be significantly more efficient than their procedural counterparts.
- Some of the more difficult aspects of compilation, such as code generation and bottom-up parsing, can be taught more easily in an O-O framework.

Acknowledgements: I would like to thank D. Kourie, K. Hemerik, and R. Watson for listening to some of these ideas and providing feedback. N. Saes proofread versions of this paper. The NWO (the Netherlands) has partly funded this research.

References

[1] Fraser, C. & Hanson, D. 1995. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings: Redwood City, California.

[2] Holmes, J. 1995. *Object-Oriented Compiler Construction.* Prentice Hall: New Jersey.

[3] Meyer, B. 1988. *Object-Oriented Software Construction.* Prentice Hall: New Jersey.

[4] Watson, B.W. 1996 (forthcoming). *The Science of Compiler Construction: An Object-Oriented Approach.*